УДК 004.75

ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ МАСШТАБУВАННЯ АРХІТЕКТУРИ ХМАРНИХ ДОДАТКІВ

О. В. Стрельцов[1]
[1]*Національний університет «Одеська політехніка»*
Ю. В. Орновецький[1]
[1]*Національний університет «Одеська політехніка»*
М. О. Катріченко[1]
[1]*Національний університет «Одеська політехніка»*

***Анотація**. Розглянуто напрямки удосконалення способу динамічного балансування навантаження для дуже важких обчислень і задач з масштабуванням на хмарній інфраструктурі, який відрізняється від відомих способів. Проведено аналіз існуючих методів і алгоритмів, які забезпечують вирішення задач балансування навантаження в хмарних інфраструктурах. Розглянуто способи балансування навантаження. Розроблено модель системи розподілення навантаження на сервіси хмарної інфраструктури на основі модифікації методу динамічного балансування навантаження. Експериментально перевірені характеристик створеної моделі. Виконане порівняння доступних методів балансування навантаження. Практична реалізація результатів дослідження дозволяє зменшити час обробки завдань, що надходять з системи обміну повідомлень, та дає можливість гнучкого налаштування серверів і додатків. Проведені експериментальні дослідження показали, що час обробки завдань при використанні запропонованого способу балансування навантаження зменшується.*

***Ключові слова:** хмарні додатки, хмарна інфраструктура, сервери, балансування навантаження, система обміну повідомлень.*

Вступ

На даний момент у світі існують дуже багато хмарних сервісів, що дозволяють розгортати застосування, які мають досить гнучкий вибір налаштувань. Актуальною задачею є аналіз можливостей використання хмарних технологій в різних сферах та визначення основних принципів підвищення ефективності обробки інформації та забезпечення її надійного зберігання у хмарних сховищах [1]. Одним із напрямків підвищення ефективності хмарних обчислень є розробка систем балансування навантаження та проектування хмарних застосувань для відповідної хмарної архітектури. На даний час існує багато систем розподілення навантаження, що працюють на базі різних алгоритмів [2]. Але наразі не запропоновано такого алгоритму, який міг би балансувати навантаження між різними серверами у складних та досить нестандартних обчислювальних задачах (наприклад одночасний рендеринг великої кількості фотографій з високою роздільною здатністю).

Найголовніша проблема подібних хмарних архітектур – це розгортання додаткових серверів для вирішення проблеми навантаження великою кількістю запитів від користувачів. Наприклад, такий процес, як рендеринг фотографій потребує дуже великої кількості ресурсів від серверів, який обробляють запити на рендеринг.

Якщо для рішення такої задачі використовувати монолітну архітектуру (з єдиним сервером), то її використання призведе до перевантаження і, як наслідок, неприпустиме збільшення часу обробки запиту користувача.

Для рішення цієї проблеми можливо використання мікросервісної архітектури, тобто розбиття її на сервіси, що обробляють різні логічні частини застосування. Але в цьому випадку виникає проблема налагодження зв'язку між сервісами при специфічній архітектурі застосування, а також проблема зі зберіганням даних у базі даних [3].

Якщо розглядати для рішення складної обчислювальної задачі SOA-архітектуру, то також можливо розбити її на сервіси, але це призведе до суттєвого ускладнення архітектури та виникнення нової проблеми із розгортанням додаткових серверів для того, щоб впоратись з великою кількістю запитів від користувачів. Але це збільшить витрати на утримування додаткових обчислювальних ресурсів [4,5].

Метою дослідження є підвищення ефективності системи розподілу навантаження на сервісах хмарної інфраструктури, що обмінюються даними між собою за допомогою системи обміну повідомленнями. Для досягнення поставленої мети необхідно вирішити наступні задачі:

1. Провести аналіз існуючих методів і алгоритмів, які забезпечують вирішення задач

балансування навантаження в хмарних інфраструктурах.
2. Розглянути існуючі способи балансування навантаження, їх переваги та недоліки.
3. Розробити модель системи розподілення навантаження на сервіси хмарної інфраструктури.
4. Експериментально перевірити характеристики створеної моделі системи розподілення навантаження.
5. Виконане порівняння ефективності використання доступних методів балансування навантаження.

1. Аналітичний огляд

1.1 Монолітна архітектура

Монолітна архітектура - це традиційна модель програмного забезпечення, яка є єдиним модулем, що працює автономно і незалежно від інших додатків. Монолітом часто називають щось велике і неповоротке, і це добре характеризує монолітну архітектуру для проектування програмного забезпечення. Монолітна архітектура — це окрема велика обчислювальна мережа з єдиною базою коду, де об'єднані всі бізнес-завдання. Щоб внести зміни в таку програму, необхідно оновити весь стек через базу коду, а також створити та розгорнути оновлену версію інтерфейсу, що знаходиться на стороні сервісу. Це обмежує роботу з оновленнями та потребує багато часу.

Моноліти зручно використовувати на початкових етапах проектів, щоб полегшити розгортання та не витрачати надто багато зусиль для керування кодом. Це дозволяє одразу випускати все, що є в монолітному додатку. Схематично монолітна архітектура зображена на рис. 1.

Рис.1. Монолітна архітектура

До недоліків монолітної архітектури можна віднести такі особливості.

- зниження швидкості розробки: великий монолітний додаток ускладнює та уповільнює розробку;
- погана масштабованість: неможливо масштабувати окремі компоненти;
- низька надійність: помилка в одному модулі може вплинути на доступність програми.
- перешкоди запровадження технологій: будь-які зміни в інфраструктурі чи мові розробки впливають на додаток цілком, що часто призводить до збільшення вартості та тимчасових витрат;
- повторне розгортання: при внесенні невеликої зміни потрібно повторне розгортання всього монолітного додатку.

1.2 Мікросервісна архітектура

Мікросервісна архітектура (або просто «мікросервіси») є методом організації архітектури, заснований на ряді служб, що незалежно розгортаються. Ці служби мають власну бізнес-логіку та базу даних з конкретною метою. Оновлення, тестування, розгортання та масштабування виконуються всередині кожної служби. Мікросервіси розбивають великі зав-дання, характерні для конкретного бізнесу, на кілька незалежних баз коду. Мікросервіси не знижують складність, але вони роблять будь-яку складність видимою і більш керованою, поділяючи завдання на дрібніші процеси, які функціонують незалежно один від одного і роблять внесок у загальне ціле [6].

Використання мікросервісів часто тісно пов'язане з DevOps, оскільки вони лежать в основі методики безперервного постачання, яка дозволяє командам швидко адаптуватися до вимог користувачів. Схематично монолітна архітектура зображена на рис. 2.

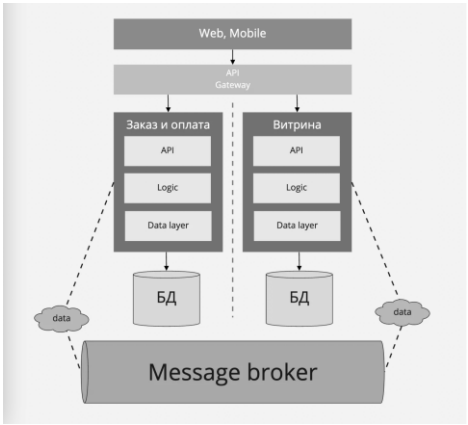


Рис.2. Мікросервісна архітектура.

Мікросервіси дають такі переваги:

- гнучкість: використання гнучких методологій розробки програмного забезпечення із безперервним розгортанням;
- гнучке масштабування: коли мікросервіс досягає граничного навантаження, можна швидко виконати розгортання нових екземплярів служби у супутньому кластері та знизити навантаження;
- безперервне розгортання: наявність регулярних та прискорених циклів релізу програмного забезпечення;
- легкість обслуговування та тестування: можливість експериментування з новими функціями з поверненням у разі необхідності до попередньої версії, спрощення оновлення коду та прискорення випуску нових функцій на ринок, спрощення пошуку та виправлення помилок та дефектів;
- незалежне розгортання: мікросервіси є окремими модулями, тому з ними можна легко і швидко виконувати незалежне розгортання окремих функцій;
- гнучкість технологій: у разі використання архітектури мікросервісів розробники можуть вибирати інструменти з урахуванням їх переваг;
- висока надійність: розгортання змін для конкретної служби не впливає на інші компоненти [7].

2. Постановка задачі.

На даний момент великим попитом користуються хмарні сервіси, які працюють із 3D-графікою (наприклад, рендеринг фотографій), монтуванням або трансляцією відео, комп'ютерними іграми і т. д. Найчастіше вони працюють через веб-браузер та надають доступ до своїх функцій, виконання яких зазвичай займає дуже багато часу для користувача (наприклад 2-3 хв). Вони використовують доволі потужні сервера. Під час розробки серверне застосування таких сервісів зазвичай не перших кроках представляє собою монолітну архітектуру.

На рис. 3 наведено схематичний приклад архітектури таких сервісів на прикладі сервісу для рендерингу 3D-турів.

На початку розробки таких проектів використовується компромісний підхід "Monolith First". Мікросервісна архітектура має важливе значення для гнучкої розробки, але використання її має значну перевагу лише для більш складних систем. Управління набором сервісів, значно сповільнює розробку, що свідчить про перевагу використання моноліту для більш простих систем [8].

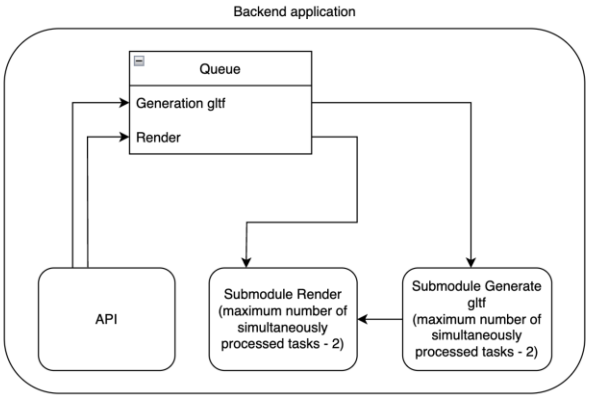


Рис. 3. Схема роботи сервісу для рендерингу 3D-турів

Це надає вагомого аргументу стратегії розробку з використанням на її початку монолітного архітектурного стилю, навіть якщо згодом проект отримає користь від архітектури мікросервісів. У такому випадку моноліт слід розробляти ретельно, звертаючи увагу на модульність у програмному забезпеченні, як на межах API, так і на збереженні даних. При збережені модульності всередині моноліту, перехід до мікросервісної архітектури є відносно простою операцією [9,10].

Отже такий підхід, відомій під назвою "Monolith First", дозволяє використати всі переваги монолітної архітектури на початку розробки системи. Наступна перевага проявляється коли з'являється вимога до використання сторонньої технології для зберігання та опрацювання даних. У такому випадку підхід дозволить швидко перейти від моноліту до мікросервісної архітектури.

Постановка задачі

3. Дослідження сучасних методів архітектурного проектування високонавантажених хмарних сервісів

Найголовнішою проблемою даної групи програмних продуктів є саме їх монолітна архітектура і високе споживання ресурсів серверів при виконанні задач таких сервісів. Отож при великій завантаженості системи виконання цих задач займає дуже багато часу, а при критичній завантаженості серверу призводить до його перезапуску, що є дуже критичним моментом для кінцевого продукту.

Використання черги завдань для таких запитів не вирішує проблему, оскільки оптимальний час виконання для користувачів виходить лише тоді, коли обробляються максимум 2 задачі на одному сервісі та 2 задачі на іншому сервісі одночасно. А це у свою чергу означає, що такий

архітектурний підхід не здатен працювати при великій кількості користувачів, оскільки чим більше одночасних запитів надходить до серверу, тим довше користувачі очікують своєї черги на виконання свого запиту.

Очікуваний час на виконання завдання буде визначатися як:

$$t = n / 2 \cdot 1.5, \qquad (1)$$

де n – кількість одночасних запитів на сервер.

Наприклад, на сервер одночасно надходять 20 запитів на рендеринг, але одночасно сервер обирає всього два, а наступні чекають своєї черги. Саме виконання 2-ох одночасних процесів рендерингу в середньому займає близько 1.5 хвилини. Отже, останньому користувачу, який дав запит, до-ведеться чекати близько 15 хвилин. І чим більше запитів, тим більший час очікування.

Для вирішення таких проблем підходить саме мікросервісна архітектура, яка базується на моделі асинхронних повідомлень.

При всій поширеності і зрозумілості підходу REST, у нього є важливе обмеження, а саме: він синхронний і, отже, блокуючий. Забезпечити асинхронність можна, але це робиться по-своєму в кожному застосуванні. Тому в деяких мікросервісних архітектурах можуть використовуватися черги повідомлень, а не модель REST – запит/відповідь.

Сервіс A (рис. 4) може синхронно викликати сервіс C, який потім буде асинхронно зв'язуватися з сервісами B і D за допомогою черги повідомлень.

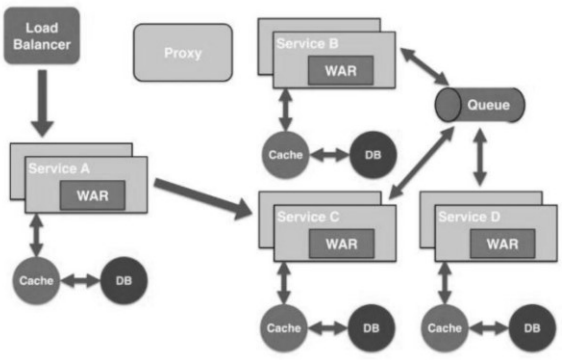


Рис. 4. Схема моделі «Асинхронні повідомлення»

Комунікація між сервісами A та C може бути асинхронною, скажімо, з використанням вебсокетів; так досягається бажана масштабованість. Комбінація моделі REST (запит/відповідь) та обміну повідомленнями (видавник/підписник) також можуть використовуватися для досягнення поставлених цілей.

3.1 Вибір моделі комунікації для мікросервісів

Перейдемо до асинхронного обміну даними на основі подій. Розглянемо два основні моменти: спосіб надання мікросервісами подій і спосіб визначення споживачами моменту настання тієї або іншої події. Традиційно брокери повідомлень, (такі, як RabbitMQ), намагаються охопити відразу обидві задачі.

Постачальники використовують API для публікації події брокеру. Брокер опрацьовує підписки, дозволяючи споживачам отримати інформацію при виникненні тієї чи іншої події. Такі брокери можуть навіть обробляти стан споживачів, наприклад сприяючи відстеженню того, які повідомлення вони бачили раніше.

Будь-яка кількість постачальників може відправляти повідомлення в одну чергу, також будь-яка кількість підписників може отримувати повідомлення з однієї черги. Підписник - програма, яка приймає повідомлення. Зазвичай, підписник знаходиться в стані очікування повідомлень.

Такі системи, як правило, розробляються з можливостями масштабування. Можливо, доведеться поплатитися ускладненням процесу розгортання, оскільки для розробки і тестування сервісів може знадобитися запуск ще однієї системи. Для збереження працездатності цієї інфраструктури можуть також знадобитися додаткові машини і наявність певно-го досвіду. Але якщо вийде справитися з усіма труднощами, це може стати дуже ефективним способом реалізації слабо пов'язаних архітектур, керованих подіями.

3.2. Масштабування архітектури

Кластер серверів (Server Cluster) — це певна кількість серверів, об'єднаних у групу та утворюючих єдиний ресурс. Дане рішення дозволяє істотно збільшити надійність і продуктивність системи.

Згруповані в локальний набір кілька комп'ютерів можна назвати апаратним кластером, однак, суть цього об'єднання — підвищення стабільності та працездатності системи за рахунок єдиного програмного забезпечення під керуванням модуля (Cluster Manager).

Загальна логіка кластера серверів створюється на рівні програмних протоколів і дає можливість:

1. управляти вільною кількістю апаратних засобів за допомогою одного програмного модуля;
2. додавати та вдосконалювати програмні та апаратні ресурси, без встановлення системи та масштабних архітектурних перетворень;
3. забезпечувати безперебійну роботу системи, при виході з ладу одного або кількох серверів;
4. синхронізувати дані між серверами — одиницями кластера;
5. ефективно розподіляти клієнтські запити по серверам;
6. використовувати загальну базу даних кластера.

По суті, головною задачею кластера серверів є виключення простої системи. В ідеалі, будь-який інцидент, пов'язаний із зовнішнім втручанням або внутрішнім зв'язком з робочим ресурсом, повинен залишатися незазначеним для користувача.

Для сервісів з високим навантаженням і з малою кількістю паралельних задач найкраща модель масштабування буде на основі виконавців. Дана модель також добре працює при пікових навантаженнях, де в міру зростання потреб можуть запускатися додаткові екземпляри для відповідності вхідного навантаження. Поки сама черга робіт буде зберігати стійкість, ця модель може використовувати масштабування для підвищення як пропускної здатності робіт, так і відмовостійкості, оскільки стає простіше впоратися з впливом відмовив (або відсутнього) виконавця. Робота потребує більше часу, але нічого при цьому не втратиться.

4. Результати експериментальної перевірки роботи моделі системи

Модель системи розподілення навантаження побудована з використанням наступних засобів і технологій:

- мова програмування Python;
- система обміну повідомленнями RabbitMQ;
- хмарна платформа Amazon Web Services (AWS);
- API-фреймворки Django Rest Framework і Celery;
- об'єктно-реляційна система управління базами даних PostgreSQL.

Проведено аналіз результатів витраченого часу на виконання високонавантажених задач.

Час виконання задач за підходом "Monolith First" представлено у таблиці 1.

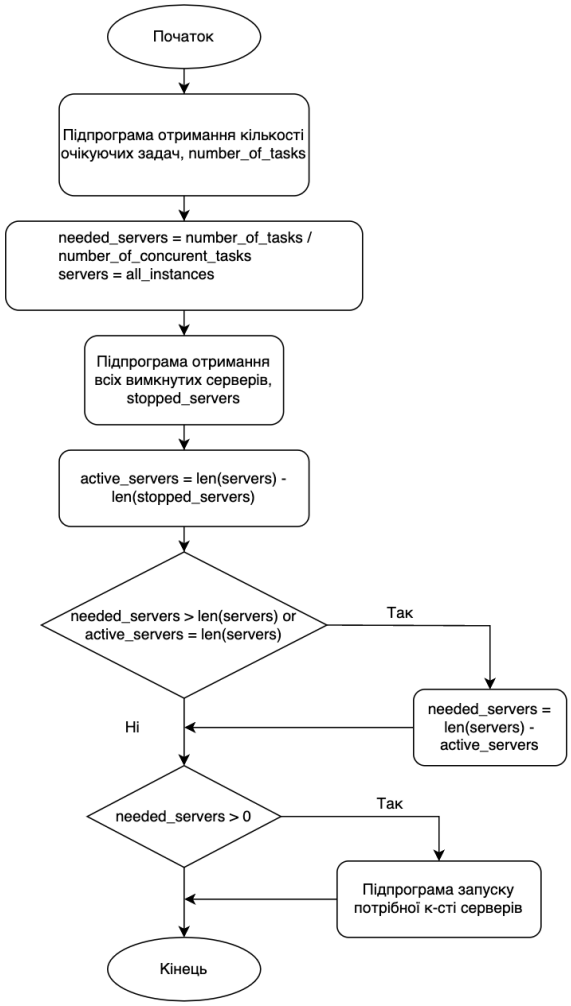


Рис. 5. Алгоритм запуску додаткових серверів у випадку перенавантаження

Таблиця 1.
Час виконання задач за підходом "Monolith First"

Кількість задач	Одночасне виконання	Час на виконання задачі на навантаження процесору, с	Час на виконання задачі на навантаження Відеокарти, с
1	-	65	22
1	+	72	27
2	+	76	46
5	+	217	119
7	+	293	165
10	+	380	230
15	+	604	349

Із таблиці 1 вище видно, що чим більше одночасних задач обробляє сервіс, тим більший час очікування для кінцевого користувача. З цього

випливає, що така модель погано підходить для високонавантажених сервісів.

Час виконання задач за підходом мікросервісної архітектури представлено у таблиці 2.

Таблиця 3.2

Кількість задач	Одночасне виконання	Час на виконання задачі на навантаження процесору, с	Час на виконання задачі на навантаження Відеокарти, с
1	-	65	22
1	+	72	27
2	+	76	46
5	+	80	43
7	+	74	41
10	+	84	45
15	+	76	50

Із даних результатів можна сказати, що модель мікросервісної архітектури з розробленою системою розподілу навантаження на сервіси хмарної інфраструктури добре підходить для високонавантажених сервісів, оскільки незалежно від кількості отриманих задач, час їх виконання залишається незмінним.

ВИСНОВКИ

В даному досліджені розглянуто основні концепції побудови додатків на базі мікросервісної архітектури. Проведено аналіз основних особливостей, переваг та недоліків даного підходу до побудови програмних систем, порівняно з класичним монолітним рішенням. Монолітна архітектура дуже добре розв'язує свої задачі, але із зростанням складності вона вже не може якісно вирішувати свої функції, тому розвинулися сервіс-орієнтовані архітектури, прикладом якої є мікросервісна архітектура.

Детально розглянуто основні принципи проектування та розгортання даних систем, основні шаблони інтеграції мікросервісів, технології комунікації. В тестовому прикладі було продемонстровано працездатність вищенаведених концепцій.

Також було розглянуто ключові компоненти для побудови мікросервісних систем, які утворюють інфраструктурний рівень та надають необхідну гнучкість всій системі. До даних компонентів відносяться: сервіс єдиного входу, сервіс відкриття, балансувальних навантаження, автоматичний вимикач. Дані шаблони було перевірено на існуючих відкритих реалізаціях.

Як результат всіх проведених досліджень, було запропоновано методику розробки мікросервісних додатків для високонавантажених систем. Для реалізації даної системи була використана мова програмування Python, а також інструменти для забезпечення інтеграції та внутрішні бібліотеки, які надають змогу розгортати мережеві розподілені додатки.

Підсумовуючи, можна стверджувати, що мікросервісна архітектура має право на існування, але не претендує стати монополістом для побудови інформаційних систем. Більше того, мікросервіси не рекомендовано створювати без глибокого попереднього аналізу предметної області та чіткого виділення обмежених контекстів, а також пропонується створювати мікросервіси на базі існуючого моноліту.

Дана архітектура є новою та перспективною у сучасному проектуванні інформаційних систем та, цілком можливо, її використання набуде масовий характер.

Список використаних джерел

1. Mell, Peter and Grance, Timothy. The NIST Definition of Cloud Computing – 2011
2. Балансування навантаження [Електронний ресурс] – Режим доступу: www / URL : https://aws.amazon.com/ru/what-is/load-balancing/
3. Порівняння мікросервісної та монолітної архітектур [Електронний ресурс] – Режим доступу: www / URL : https://www.atlassian.com/ru/microservices/microservices-architecture/microservices-vs-monolith
4. SOA architecture [Електронний ресурс] – Режим доступу: www / URL : https://rubygarage.org/blog/monolith-soa-microservices-serverless
5. Мікросервіси vs SOA [Електронний ресурс] – Режим доступу: www / URL : https://scoutapm.com/blog/soa-vs-microservices#h_84112960013771642610665654
6. Introduction to microservices.\ [Електронний ресурс] – Режим доступу: www / URL : https://nginx.com/blog/introduction-to-microservices/
7. Fowler M. Monolith First [Електронний ресурс] / Martin Fowler. – 2015. –
8. Режим доступу до ресурсу: https://martinfowler.com/bliki/MonolithFirst.html#footnotetypical-monolith
9. Ньюмен С. Создание микросервисов / Ньюмен С. – СПб.: Питер, 2017– 304 с.
10. Мікросервісні патерни проектування [Електронний ресурс] – Режим доступу: www /

URL : https://habr.com/ru/company/piter/blog/275633/

References

1. Mell, Peter and Grance, Timothy. The NIST Definition of Cloud Computing – 2011
2. Load balancing [Electronic resource] – Access mode: www / URL: https://aws.amazon.com/ru/what-is/load-balancing/
3. Comparison of microservice and monolithic architectures [Electronic resource] – Access mode: www / URL: https://www.atlassian.com/ru/microservices/microservices-architecture/microservices-vs-monolith
4. SOA architecture [Electronic resource] – Access mode: www / URL: https://rubygarage.org/blog/monolith-soa-microservices-serverless
5. Microservices vs SOA [Electronic resource] - Access mode: www / URL: https://scoutapm.com/blog/soa-vs-microservices#h_84112960013771642610665654
6. Introduction to microservices.\ [Electronic resource] - Access mode: www / URL: https://nginx.com/blog/introduction-to-microservices/
7. Fowler M. Monolith First [Electronic resource] / Martin Fowler. – 2015. –
8. Resource access mode: https://martinfowler.com/bliki/MonolithFirst.html#footnotetypical-monolith
9. Newman S. Creation of microservices / Newman S. – St. Petersburg: Peter, 2017– 304 p.
10. Microservice server cluster [Electronic resource] – Access mode: www / URL: https://www.stekspb.ru/autsorsing-it-infrastructure/it-glossary/server-cluster/
11. Microservice design patterns [Electronic resource] – Access mode: www / URL: https://habr.com/ru/company/piter/blog/275633/

INCREASING THE EFFICIENCY OF SCALING CLOUD APPLICATIONS ARCHITECTURE

Oleh Streltsov[1], Yuriy Ornovetsky[1], Mykhailo Katrichenko[1]
[1]Odessa National Polytechnic University

Abstract. *The areas of improvement of the method of dynamic load balancing for very heavy calculations and tasks with scaling on cloud infrastructure, which differ from known methods, are considered. An analysis of existing methods and algorithms that provide solutions to load balancing problems in cloud infrastructures was carried out. Methods of load balancing are considered. A model of the load distribution system for cloud infrastructure services based on a modification of the dynamic load balancing method has been developed. Experimentally verified characteristics of the created model. Comparison of available load balancing methods. The practical implementation of tracking results allows to reduce the processing time of tasks that come from the messaging system and enables flexible configuration of servers and applications. The conducted experimental studies showed that the processing time of operations decreases when using the proposed method of load balancing.*

Keywords: *cloud applications, cloud infrastructure, servers, load balancing, message exchange system.*

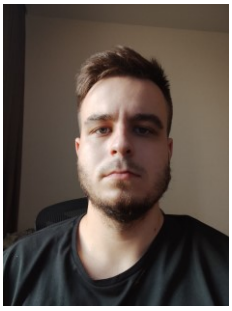
Отримано 05.10.2023

Стрельцов Олег Васильович, Національний університет «Одеська політехніка» кандидат технічних наук, доцент, доцент кафедри комп`ютерних систем. Просп. Шевченка, 1, Одеса, Україна, E-mail: streltsov.o.v@op.edu.ua, тел. +38-048-705-84-73
Oleh Streltsov, Odesa National Polytechnic University, Candidate of Technical Sciences, Associate Professor, Assistant Professor of the Computer Systems Department, Shevchenko ave., 1, Odesa, Ukraine, e-mail: streltsov.o.v@op.edu.ua, ph. +38-048-705-84-73

ORCID ID: 0000-0002-4691-5703

Орновецький Юрій Васильович, Національний університет «Одеська політехніка», аспірант кафедри комп`ютерних систем. Просп. Шевченка, 1, Одеса, Україна, E-mail: y.ornovetskyi@gmail.com, тел. +38-063-050-4833
Yuriy Ornovetsky, Odesa National Polytechnic University, graduate student of the Department of Computer Systems.. Shevchenko ave, 1, Odesa, Ukraine, E-mail: y.ornovetskyi@gmail.com,
ph. +38-063-050-4833

ORCID ID: 0009-0006-2470-1559



Катріченко Михайло Олегович, Національний університет «Одеська політехніка», аспірант кафедри комп`ютерних систем. Просп. Шевченка, 1, Одеса, Україна, E-mail: m_katrichenko@ukr.net, тел. +38-063-302-6906.
Mykhailo Katrichenko, Odesa National Polytechnic University, graduate student of the Department of Computer Systems.. Shevchenko ave, 1, Odesa, Ukraine, E-mail: m_katrichenko@ukr.net,
ph. +38-063-302-6906.

ORCID ID: 0009-0004-2251-5600